Chapter 9

Advanced Assembly Language Programming

Pointers to pointers

- Occur frequently with linked lists
- Require double dereferencing (to follow the two pointers)

Review of single pointer

FIGURE 9.1

```
#include <iostream>
using namespace std;

void f(int *p)                          // p receives the address of x
{
      *p = 3;                           // x is assigned 3
}
void main()
{
      int x;
      f(&x);                            // &x is an int *
      cout << "x = " << x << endl;      // outputs 3
}
```

Review of single pointer (using reference parameters)

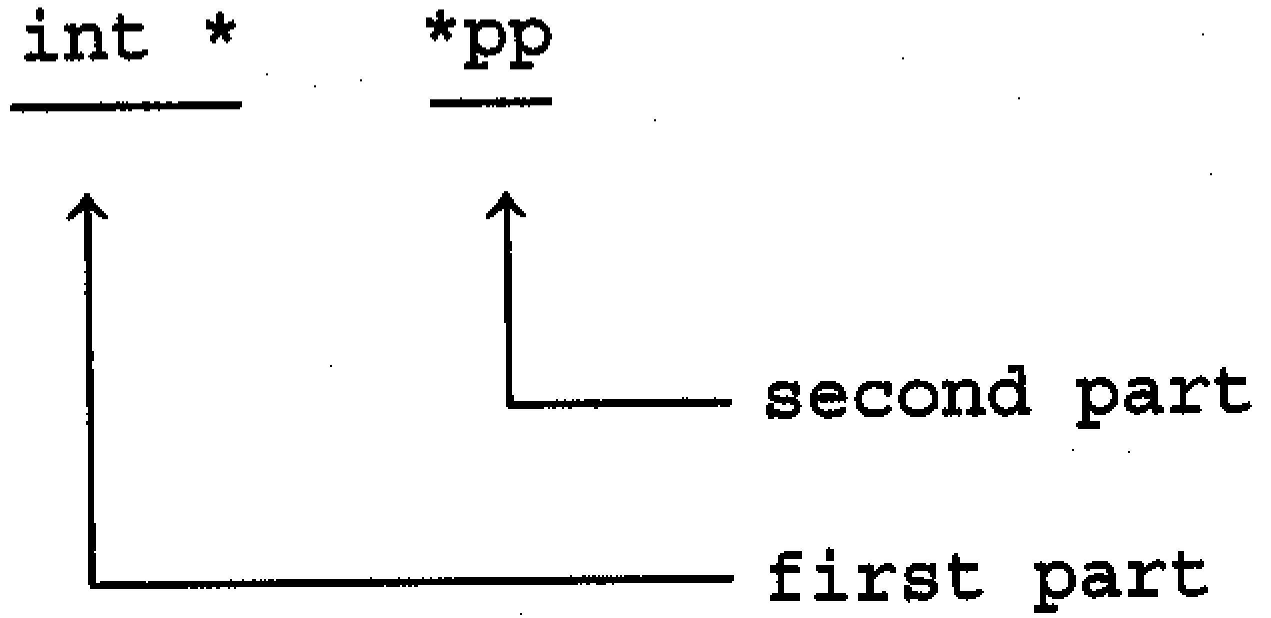
FIGURE 9.2

```
#include <iostream>
using namespace std;

void f(int &z)                      // z is a reference parameter
{
     z = 3;                         // x is assigned 3
}
void main()
{
     int x;
     f(x);                          // x itself is passed to f
     cout << "x = " << x << endl;   // outputs 3
}
```

pp is a pointer to a pointer

FIGURE 9.3

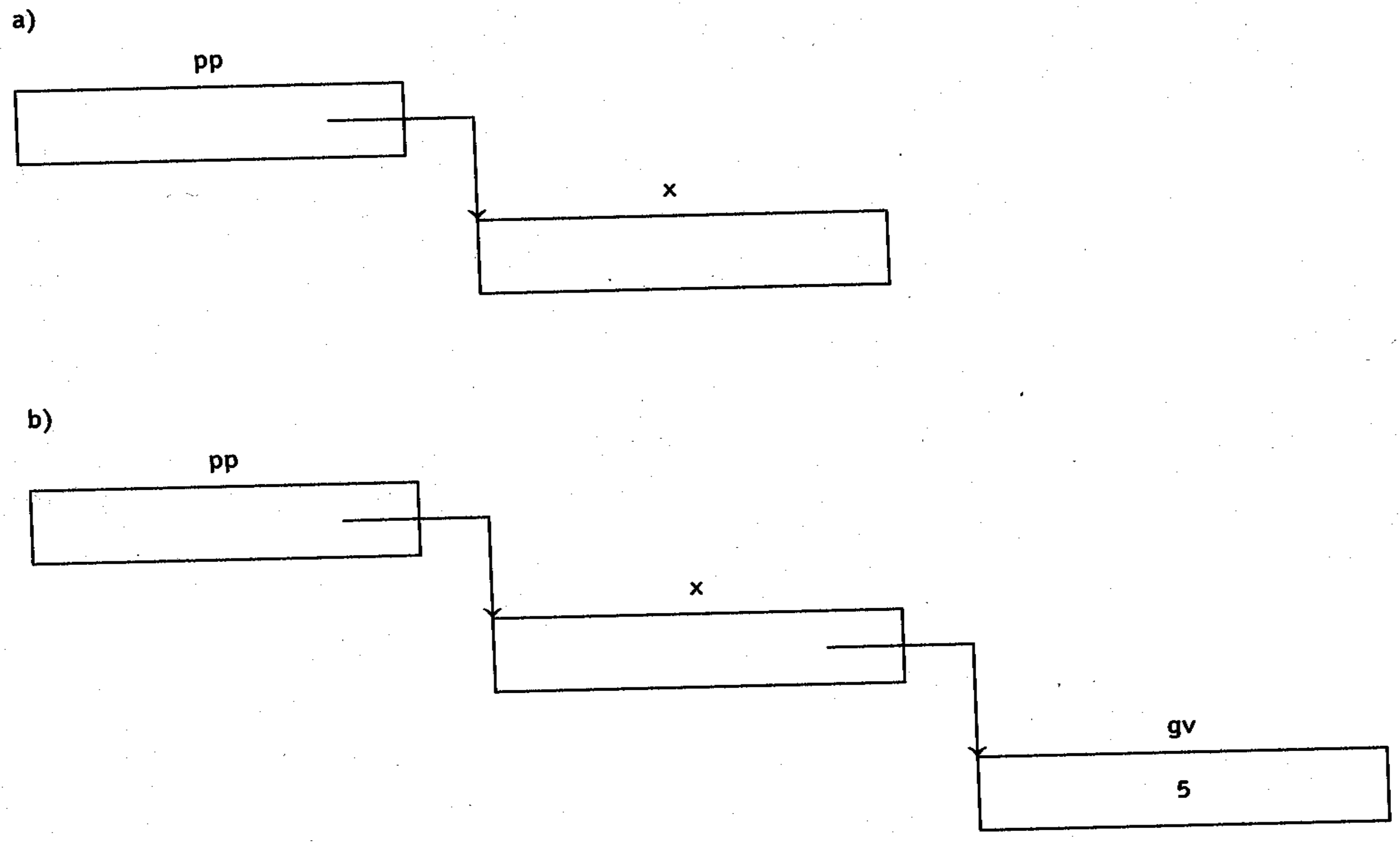
```
#include <iostream>
using namespace std;

int gv = 5;
void f(int **pp)                    // pp receives address of x
{
        *pp = &gv;                  // assigns x address of gv
        cout << **pp << endl;       // outputs contents of gv
}
void main()
{
        int *x;
        f(&x);                      // &x is an int **
        cout << *x << endl;         // outputs contents of gv
}
```



Read as “pp is a pointer (the second part)
to an int pointer (the first part)”

pp assigned address of x. Then x assigned the address of gv

FIGURE 9.4



Double asterisk in a declaration means pointer to a pointer

```
int  **pp
```

indicates pp is a pointer to a pointer

A double asterisk in executable statement specifies two dereferencing operations

```
cout << **pp << endl;
```

double dereference

Rewrite program with reference parameters

FIGURE 9.5

```
#include <iostream>
using namespace std;

int gv = 5;
void f(int *&z)                     // z is a reference parameter
{
    z = &gv;                        // x assigned address of gv
    cout << *z << endl;             // display *x
}
void main()
{
    int *x;
    f(x);                           // call by reference
    cout << *x << endl;             // display *x
}
```

Assignment to reference parameter requires one dereferencing operation. Places address of gv into what z points to (i.e., x).

```
z = &gv;
```

one dereferencing operation occurs here

cout << *z << endl;

Two dereferencing operations required:

The first implicit because z is a reference parameter.

The second explicit because of ‘*’

The pointer program and the reference parameter program translate to exactly the same assembly code, except for name mangling.

```
void f(int **p)     // @f$ppi

void f(int *&z)   // @f$rpi
```

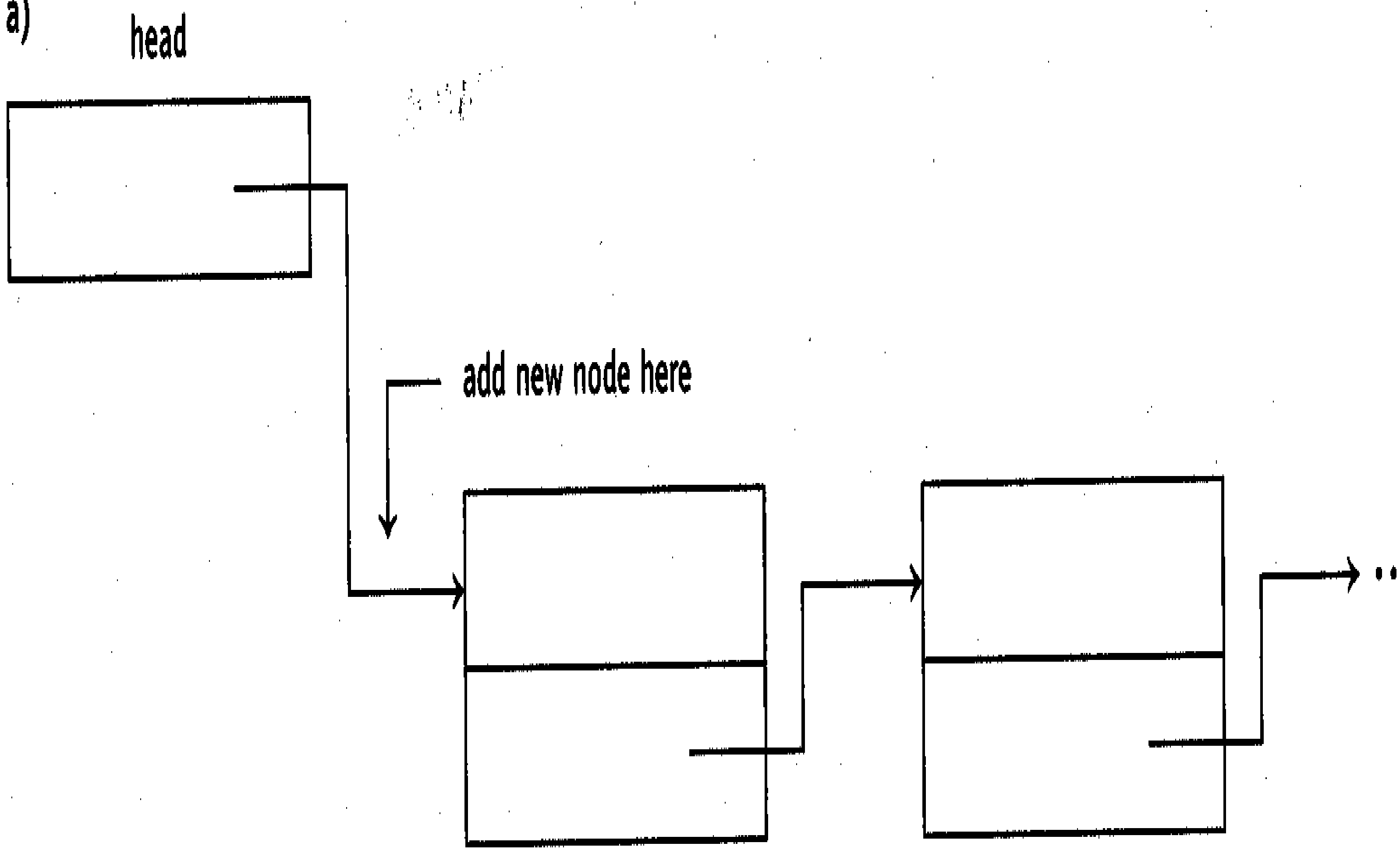
FIGURE 9.6

```
@f$rpi:                                 ; *pp = &gv;
               ldc   gv                 ; get address of gv
               push
               ldr   2                  ; get pp
               sti                      ; pop address of gv into *pp

                                        ; cout << **pp << endl;
               ldr   1                  ; get pp
               ldi                      ; load *pp
               ldi                      ; load **p
               dout                     ; display it
               ldc   '\n'               ; newline
               aout

               ret
;======================================================================
main:          aloc 1                   ; int *x;

                                        ; f(&x);
               swap
               st    @spsave
               swap                     ; restore sp
               ld    @spsave            ; get absolute address of x
               push                     ; create parameter
               call  @f$rpi
               dloc 1                   ; deallocate parameter

                                        ; cout << *x << endl;
               ldr   0                  ; get x
               ldi                      ; get *x
               dout                     ; display it
               ldc   '\n'               ; newline
               aout

               dloc 1                   ; deallocate x
               halt
gv:            dw    5
@spsave:       dw
               end   main
```

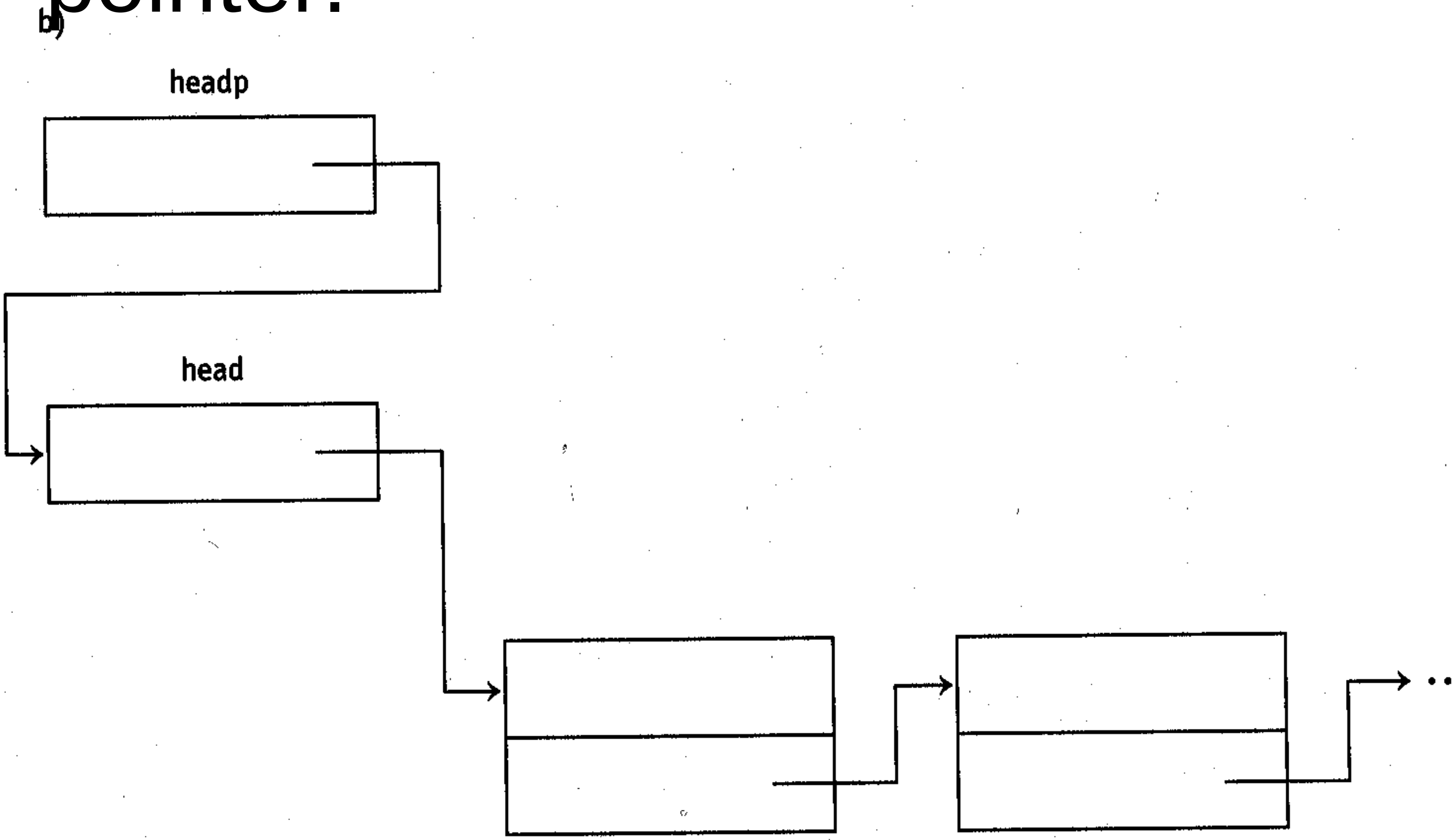
A function that adds a new node to a linked list requires a parameter that is a pointer to a pointer.

FIGURE 9.7

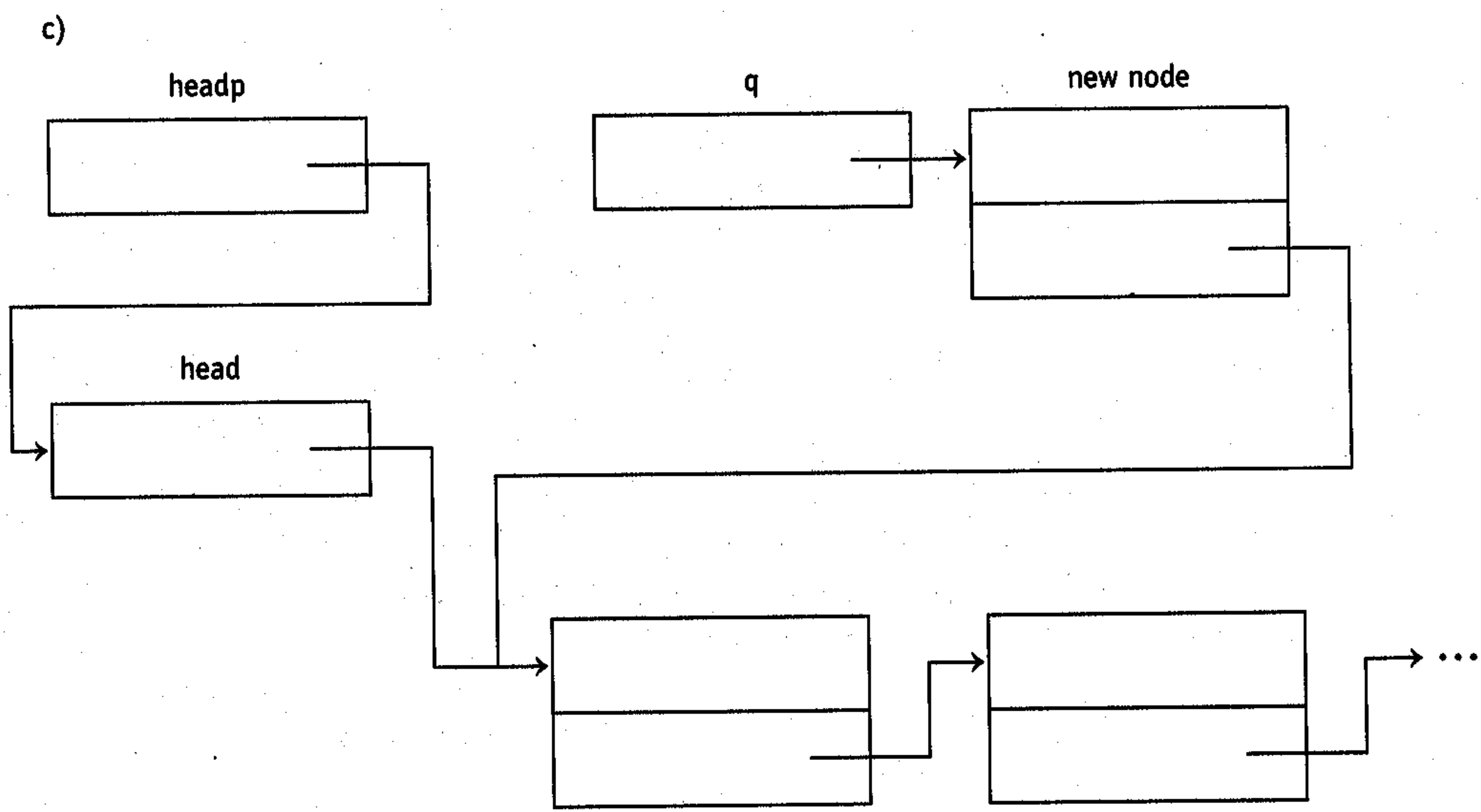
a)



Call function with parameter **headp.** **headp** is a pointer to a pointer.

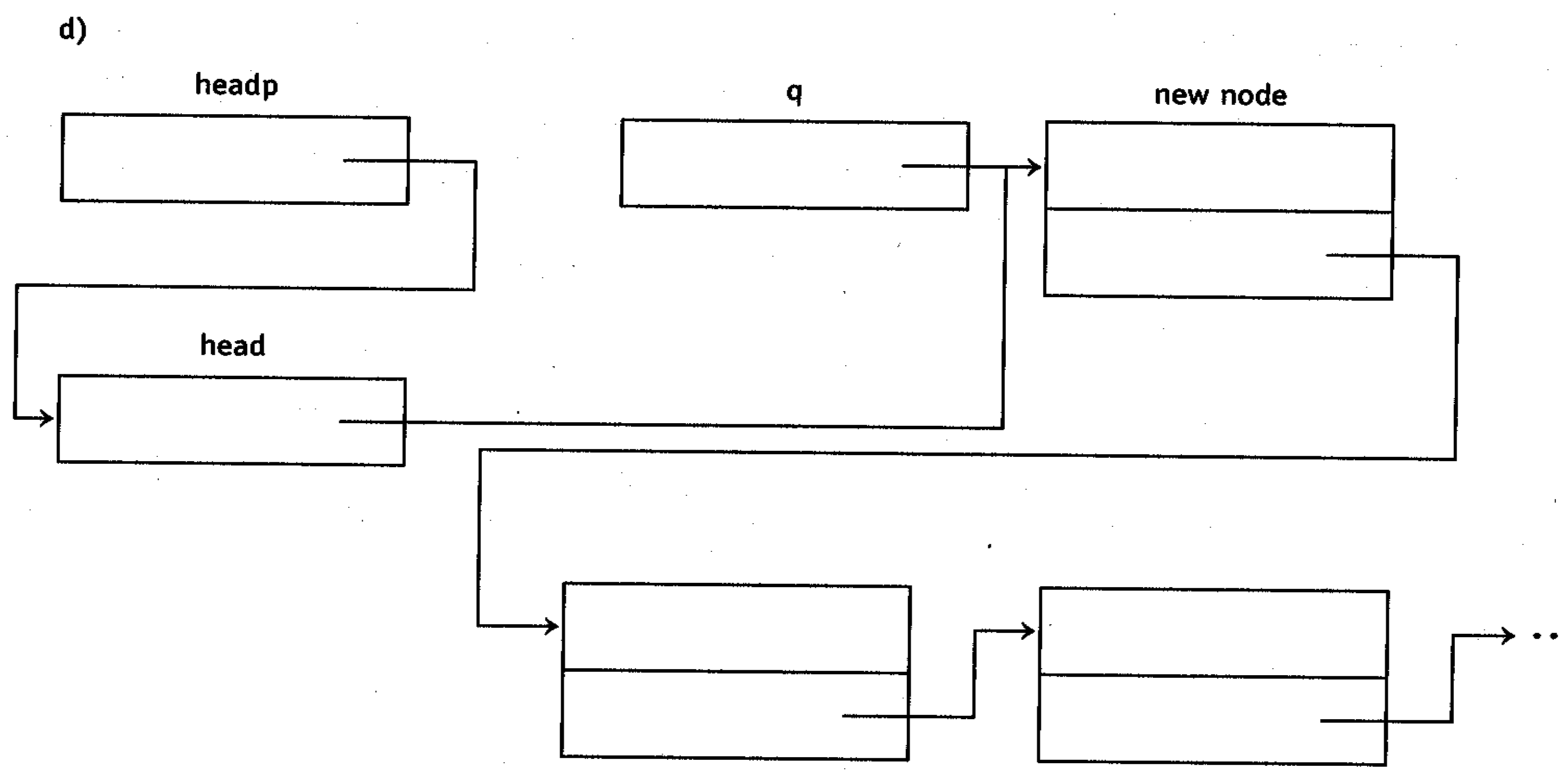


Create a new node and set its link field.



Make **head** point to new node.
This requires a pointer to **head**.
Thus, the parameter **headp** must be a pointer to a pointer.

FIGURE 9.7



Pointer to pointer version

```
void add_node(NODE **headp, int x)
{
      NODE *q;

      q = new NODE;        // create new node
      q -> data = x;       // add data to new node
      q -> link = *headp;   // head to link of new node
      *headp = q;           // get head to point to new node
}
```

Reference parameter version (same assembly code as pointer version)

```
void add_node(NODE *&head, int x)
{
  NODE * q;

  q = new NODE;
  q -> data = x;
  q -> link = head;
  head = q;
}
```

r(x + y);   // call by reference

- Can implement using an implicit local variable allocated on the stack to hold the value of x + y. This approach requires an awkward swap-st-swap sequence.
- Can implement using a temporary variable implemented with a **dw** statement. This approach does not require the swap-st-swap sequence. But there is a potential bug in this approach.

Using an implicit local variable

```
ld x          ; get x
add y         ; compute x + y (assuming y is a global variable)
push          ; store result in implicit variable
swap          ; get address of this variable
st    @spsave
swap
ld    @spsave
push          ; push address of implicit local variable
call     @r$ri
dloc 2        ; deallocate parameter and implicit variable
```

Using @temp defined with a dw

```
ld   x
add  y
st   @temp
ldc  @temp
push
call @r$ri
dloc 1
```

where **@temp** is defined with

```
@temp:     dw     0
```

For the following program, output for implicit variable implementation differs from output with @temp implementation

FIGURE 9.8

```
#include <iostream>
using namespace std;

int x = 3;
void bug(const int &m)
{
        if (m != 0)
            bug(m - 1);
        cout << m << endl;
}
void main()
{
        bug(x);
}
```

Code using temp for recursive call. Values in @temp get overlaid.

FIGURE 9.9

```
; compute value of m - 1
ldr     1               ; get the address in m
ldi                     ; dereference this address
sub     @1              ; compute value of argument

; store this value in @temp
st      @temp

; pass the address of @temp
ldc     @temp           ; get address of @temp
push                    ; create parameter (the next m)
call    @bug$ri
dloc    1               ; deallocate parameter
```

If we translate this program using the implicit variable approach, and run it, it displays

```
0
1
2
3
```

This is the correct output. If, however, we translate the same program using the `@temp` approach, we get something different:

```
0
0
0
3
```

Same problem occurs in this program

FIGURE 9.10

```
#include <iostream>
using namespace std;

int f(int a)
{
        int y;
        if (a != 0) {
            y = (a + 1) + f(a - 1);         // bug if @temp used
            return y;
        }
        return 10;

}
void main()
{
        cout << f(2) << endl;
}
```

Assembly code for line 8 in the program on the preceding slide is wrong if an @temp is used to hold (a+1) during the execution of f(a-1);

Correct code for line 8

```
ldc 1          ; get 1
addr 2         ; add a
push           ; save value of subexpr by pushing
ldr 3          ; get a
sub @1         ; subtract 1
push           ; create parameter for f
call @f$i      ; call f
dloc 1         ; remove parameter
addr 0         ; add saved value of first subexpression
dloc 1         ; remove saved value from stack
str 0          ; store in y
```

Relational and Boolean expressions

- In C++, non-zero represents true;
  0 represents false.
- A 0 or 1 value does not always have to be generated when using relational and Boolean expressions. See the following slides.

```
z = x == y;
```

we are assigning the value of the relational expression `x == y` to `z`. For this statement, a compiler must produce code that provides a value for the relational expression, which can then be assigned to `z`. Our compiler will generate code that produces either 1 (for true) or 0 (for false). The assembly code for this statement is

```
        ld   x
        sub  y
        jnz  @L4    ; jump if relational expression is false
        ldc  1      ; load true
        ja   @L5
@L4:    ldc  0      ; load false
@L5:    st   z      ; assign value to z
```

Must produce 0/1 value here also

```
f(x == y);
```

is translated to

```
        ld x
        sub y
        jnz @L6      ; jump if relational expression is false
        ldc 1        ; load true
        ja @L7
@L6:    ldc 0        ; load false
@L7:    push
        call @f$i
        dloc 1
```

Do not have to produce 0/1 value here

```
if (x == y)
   z = 10;
```

can be translated to

```
        ld x
        sub y
        jnz      @L8   ; jump if false
        ldc      10    ; assign 10 to z
        st     z
@L8:
```

Relational operators have higher precedence than Boolean operators

```
a == b || c == d
```

evaluate this subexpression first

Short-circuited evaluation

```
if (x == y || x == z)
   z = 10;
```

is

```
          ld x     ; evaluate left subexpression
          sub y
          jz @L0   ; jump immediately to assignment stmt if true
          ld x     ; evaluate right subexpression
          sub z
          jnz @L1
@L0:      ldc  10
          st   z
@L1:
```

Short-circuited evaluation needed here to ensure that i is a valid index.

```
if (i >= 0 && i  < SIZE && a[i] == 5) {
      .
      .
      .
}
```

Short-circuited evaluation needed here to ensure that a null pointer is not dereferenced.

```
if (p != NULL && *p == 5) {
        .
        .
        .
}
```

Without short-circuited evaluation, must do this

```
if (p != NULL)
    if (*p == 5) {
            .
            .
            .
    }
```

Need short-circuited evaluation here

```
while (p != NULL && *p == 5) {
        .
        .
        .
}
```

Without short-circuited evaluation, must do this

```
more = true;
while (p != NULL && more)
    if(*p == 5) {
          .
          .
          .
    }
    else
        more = false;
```

Evaluation approach can affect a computation as well as execution time and program complexity.

```
( x == y || y == f() )
```

With short-circuited evaluation, f() might not be called. This could affect the computation if f() has any side effects (such as setting global variables).

As with relational expressions, the code generated for a Boolean expressions depends on context. For example, for the statement

```
b = b1 && b2;
```

the compiler has to generate code to produce 1 (true) or 0 (false), which can then be assigned to b. However, in the statement

```
if (b1 && b2) {
.
.
.
}
```

the compiler does not have to generate code that produces the 1 and 0 values because the value of the Boolean expression is not assigned to anything.

Boolean expression evaluation in Java

Can be shorted-circuited:

    b1 && b2

or not short-circuited:

    b1 & b2

Strings in C++--two types

- C-type strings which are implemented as character arrays.
- Objects instantiated from the String class

We will consider C-type strings only.

A string constant is translated to the address of its 1st character. Thus its type is char *.

```
char ctab[10];
char *cp;
cp = ctab;
cp = “hello”;
ctab = “hello”;   // illegal
strcpy(ctab, “hello”);  // okay
```

Warning: every instance of a C-type string constant should map to a separate dw statement.

See the next slide.

FIGURE 9.11

```
#include <iostream>
using namespace std;

char *gp;
void mod_string(char *p)
{
      *p = 'X';
}
void main()
{
      mod_string("abc");
      gp = "abc";
      cout << gp << endl;      // would output Xbc if only one dw for "abc"
}
```

Don’t need multiple identical string constants in Java because string constants are immutable in Java.

Call by value-result

- Not supported by C++ or Java
- Has the efficiency of call by value but permits side effects like call by reference
- In our implementation of call by value-result, we use ‘$’ to mark a value-result parameter in a C++ program, like ‘&’ is used to mark a reference parameter in void vr(int &x) {…}

Output of the program on the next slide is

$y = 6$

$y = 6$

$y = 1$

indicating that the first (call by reference) and second (call by value-result) calls have a side effect (changing the value of y from 1 to 6). The third call is call by value.

FIGURE 9.12

```
#include <iostream>
using namespace std;

int y;
void ref(int &x)                    // & signals the reference mechanism
{
     x = x + 5;
}

void vr(int $x)                     // $ signals the value-result mechanism
{
     x = x + 5;
}

void v(int x)                       // just x signals the value mechanism
{
     x = x + 5;
}

void main()
{
     y = 1;
     ref(y);
     cout << "y = " << y << endl;
     y = 1;
     vr(y);
     cout << "y = " << y << endl;
     y = 1;
     v(y);
     cout << "y = " << y << endl;
}
```

The call-by-value function and the call-by-value-result function are translated to the same assembly code:

```
ldc   5      ; get 5
addr 1       ; add x
str   1      ; store result back into x
ret
```

Call by value and call by value-result have different calling sequences. Call-by-value calling sequence:

```
v(y);
```

is

```
ld   y
push        ; create parameter x
call @v$i
dloc 1      ; remove parameter x
```

In call by value-result, the parameter is used to effect a side effect. Call-by-value-result calling sequence:

```
vr(y);
```

is

```
ld   y
push         ; create parameter x
call @vr$mi
pop          ; remove parameter x by popping into ac reg
st   y       ; store value of parameter x in argument y
```

FIGURE 9.13

```
@ref$ri:    ldr     1               ; x = x + 5;
            ldi
            add     @5
            push
            ldr     2
            sti

            ret
;==========================================================================
@vr$mi:     ldc     5               ; x = x + 5;
            addr    1
            str     1

            ret
;==========================================================================
@v$i:       ldc     5               ; x = x + 5;
            addr    1
            str     1

            ret
;==========================================================================
cout:       ldc     @m0
            sout
            ld      y
            dout
            ldc     '\n'
            aout
            ret
;==========================================================================
main:       ldc     1               ; y = 1;
            st      y

            ldc     y               ; ref(y);
            push
            call @ref$ri
            dloc 1

            call cout               ; cout << "y = " << y << endl;

            ldc     1               ; y = 1;
            st      y

            ld      y               ; vr(y);
```

(continued)

FIGURE 9.13 (continued)

```
          push
          call @vr$mi
          pop
          st      y

          call cout             ; cout << "y = " << y << endl;

          ldc     1             ; y = 1
          st      y

          ld      y             ; v(y);
          push
          call @v$i
          dloc 1

          call cout             ; cout << "y = " << y << endl;

          halt
y:        dw      0
@5:       dw      5
@m0:      dw      "y = "
          end     main
```

Run time is longer if x is a reference parameter rather than a value-result parameter because of the dereferencing operations that have to be performed.

```
for (int i = 0; i < 30000; i++)
    x = x + 5;
```

60,000 dereferencing operations if x is a reference parameter.

Don’t really need value-result—use local variable for computations

FIGURE 9.14

```
void simvr(int *p)
{
    int x;
    x = *p;

    // body of function-use x instead of *p

    *p = x;
}
```

Call by reference and call by value do not always produce the same results.

In the program on the next slide, the output is

$y = 1$

$y = 6$

Reference and value-result not always the same

FIGURE 9.15

```
#include <iostream>
using namespace std;

int y;
void putoff(int $x)
{
     x = x + 5;                          // side effect is delayed
     cout << "y = " << y << endl;
}
void rightnow(int &x)
{
     x = x + 5;                          // side effect is immediate
     cout << "y =" << y << endl;
}
void main()
{
     y = 1;
     putoff(y);
     y = 1;
     rightnow(y);
}
```

It is difficult to implement the calling sequence for value-result when calling a function that uses the return statement. If we pop the parameter to store its value in its corresponding argument, we destroy the value returned in the ac register.

One solution: use the value in ac first, then pop the parameter.

Problem with value-result and return statement

FIGURE 9.16

```
#include <iostream>
using namespace std;

int x, y;
int add_one(int $z)
{
    return z + 1;
}
void main()
{
    x = 1;
    y = add_one(x);     // value returned is assigned to y
    cout << "y = " << y << endl;
}
```

```
ld x
push
call @add_one$mi
pop                 ; get value of z--clobbers ac register
st   x              ; store value of z in x
```

But, by doing so, it destroys the value in the **ac** register that the **add_one** function is returning. We, of course, need this return value (it has to be assigned to **y**). The fix for this problem in this case is simple: On return from **add_one**, we first use the return value in the **ac** register. We can then pop the stack to obtain the value of **z**:

```
ld   x
push
call @add_one$mi
st   y              ; use value returned in ac register
pop                 ; get value of z
st   x              ; store value of z in x
```

Another problem with value-result and the return statement is illustrated below. To handle this call, we must do steps 1, 2, and 3. Cannot simply use the return value first.

```
y = f(add_one(x), 3);
```

1. Save the return value in an implicit local variable when **add_one** returns.
2. Finish the call of **add_one** (i.e., pop the parameter and store in **x**).
3. Use the saved return value.

Here is the required code:

```
                ; start call of f
ldc  3          ; push arg 3 for call of f
push
                ; start call of add_one
aloc 1          ; create an implicit local variable
ld x
push
call @add_one$mi
str 1           ; save return value in implicit local variable
pop             ; finish call of add_one
st   x
                ; now continue the call of f
                ; we don't have to push returned value
                ; because it is already on the stack
call f
dloc 2
```

Variable-length argument lists

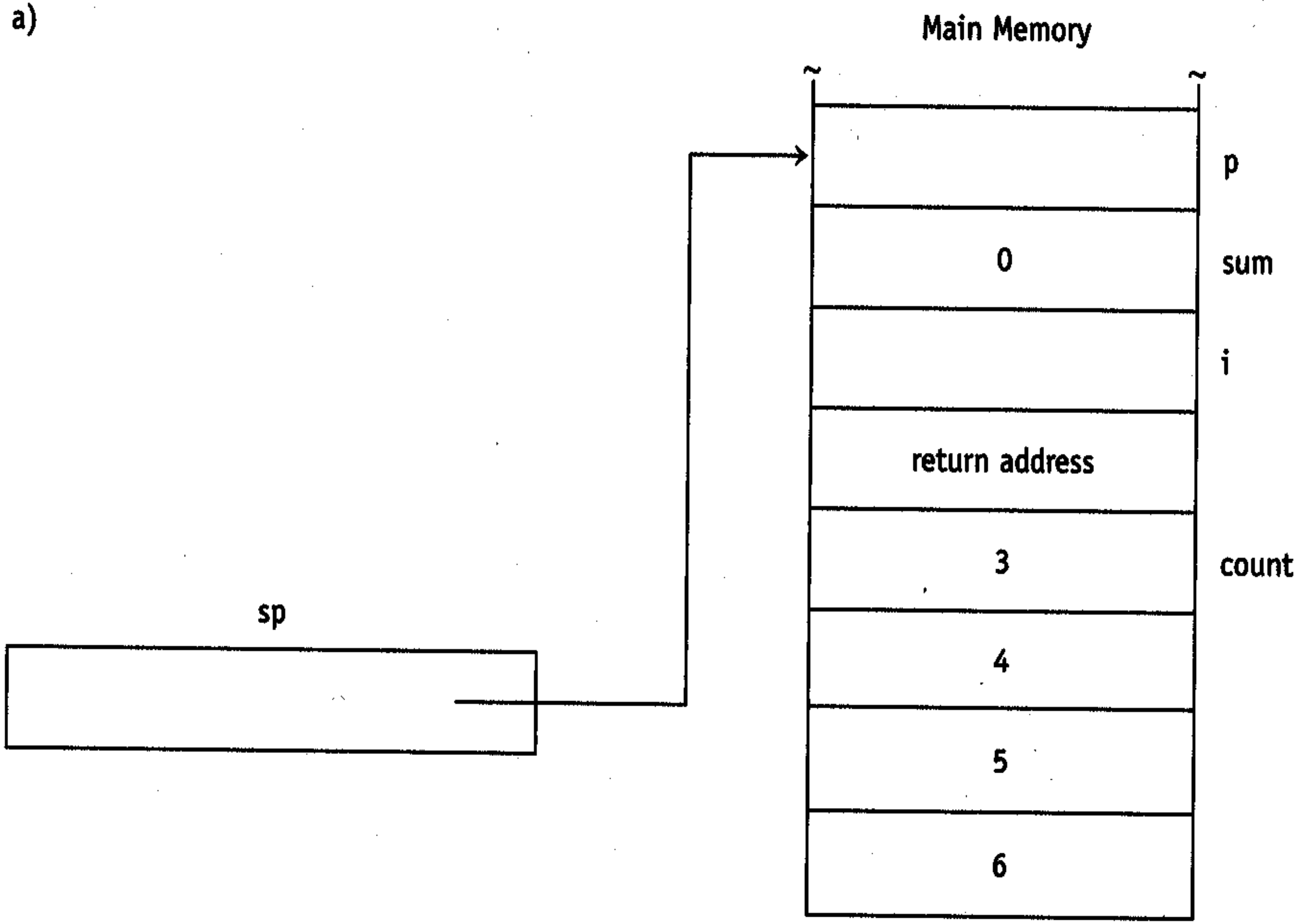
- Use ellipsis (“…”): int add(int count, …)
- First argument must indicate the number of arguments.
- The address of the second parameter given by (address of first parameter) + 1
- The compiler knows where the first parameter is (it is right above the return address).

FIGURE 9.17

```
#include <iostream>
using namespace std;

int add(int count, ...)        // ... means variable number of parameters
{
     int i, sum = 0;
     int *p;

     p = &count + 1;           // p now points to first param to be added

     for (i = 1; i <= count; i++)
     sum = sum + *p++;

     return sum;
}
//  ;==========================================================
void main()
{
     // arguments are pushed in right-to-left order
     cout << add(3, 4, 5, 6) << endl;       // outputs 15
     cout << add(2, -10, 20) << endl;       // outputs 10
     cout << add(1, 7) << endl;             // outputs 7
}
```

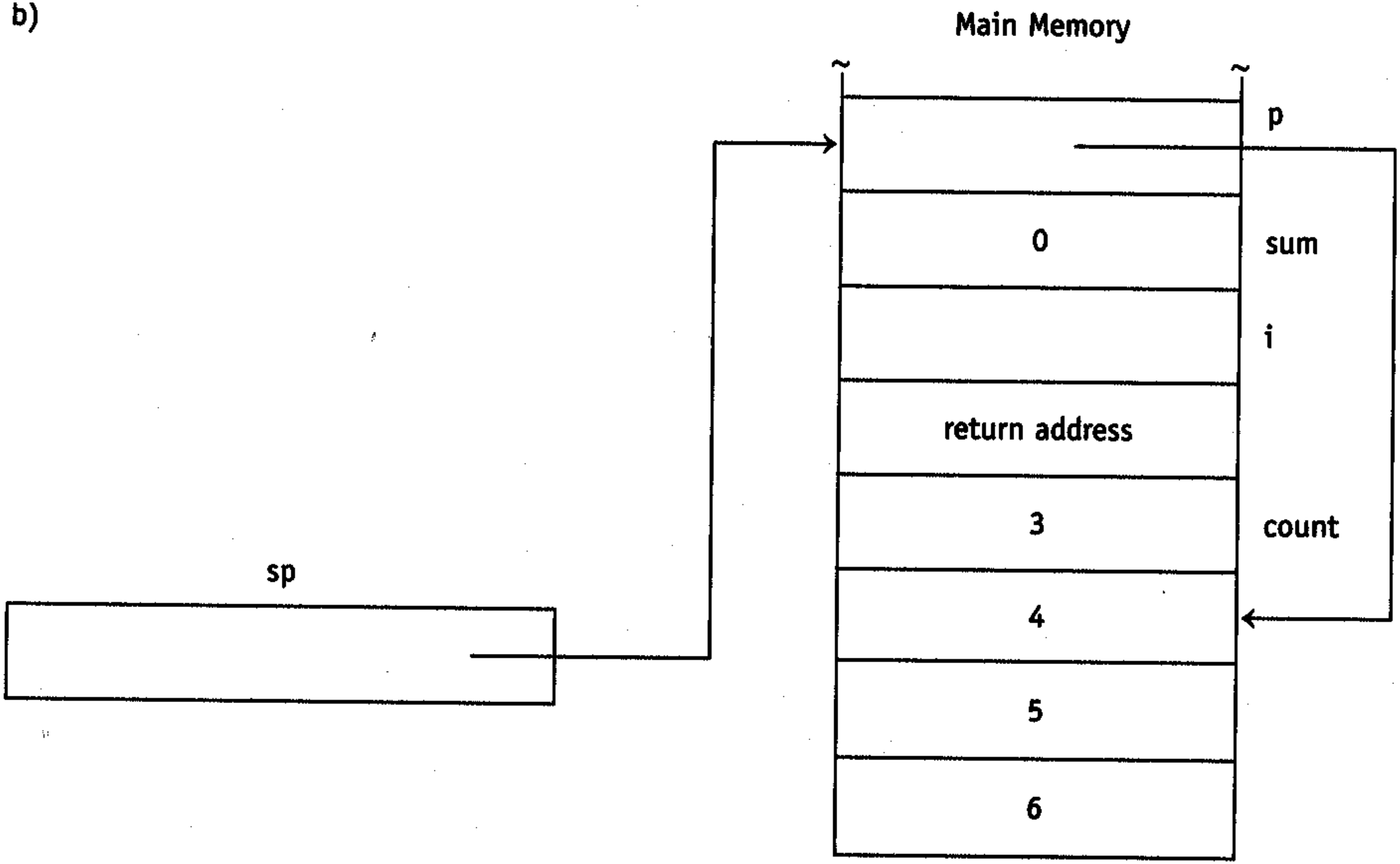
Stack on entry.
count is right above the return address

FIGURE 9.18 a)



p assigned address of 2nd parameter. Then use p to access remaining parameters.

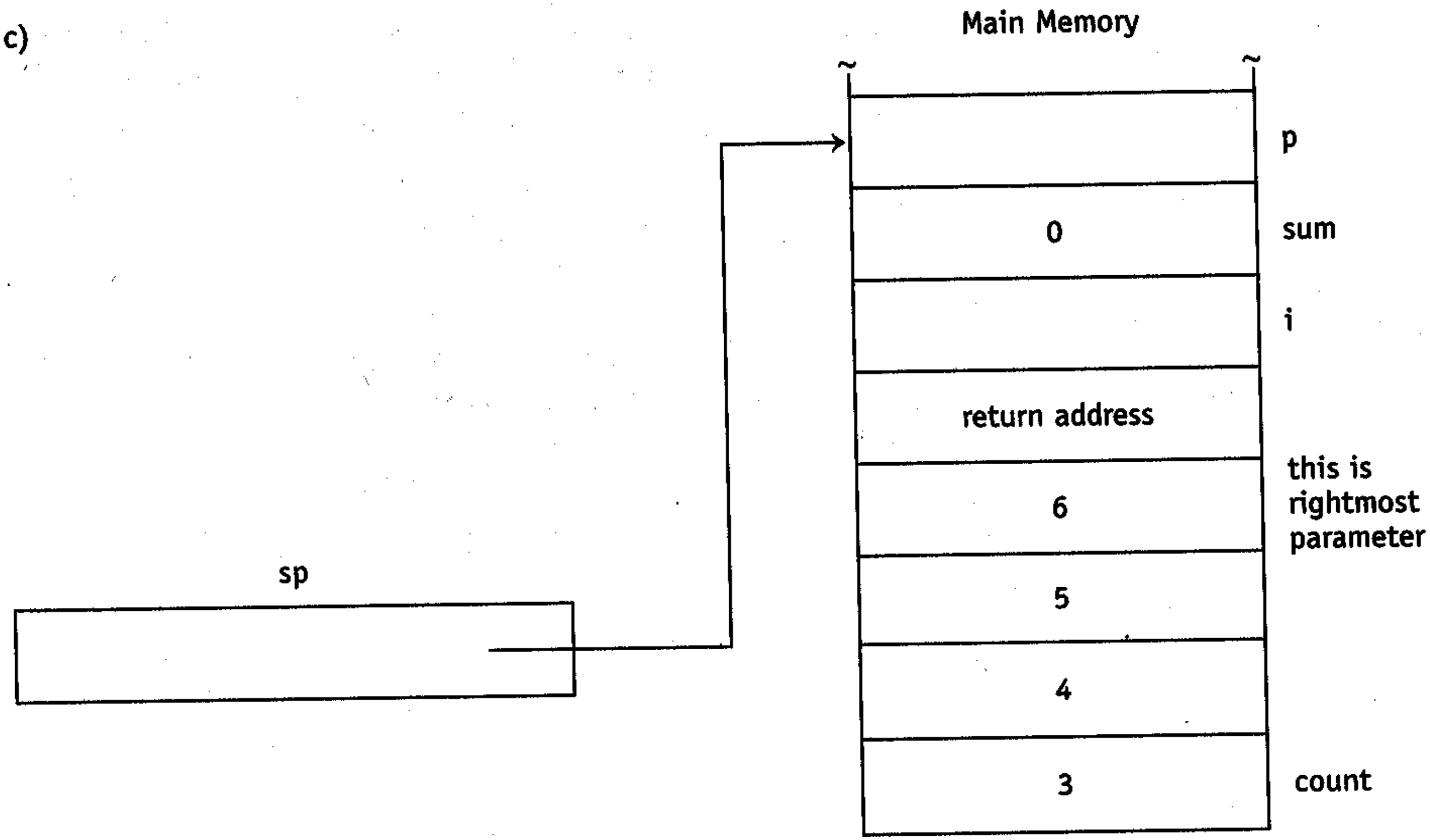
b)



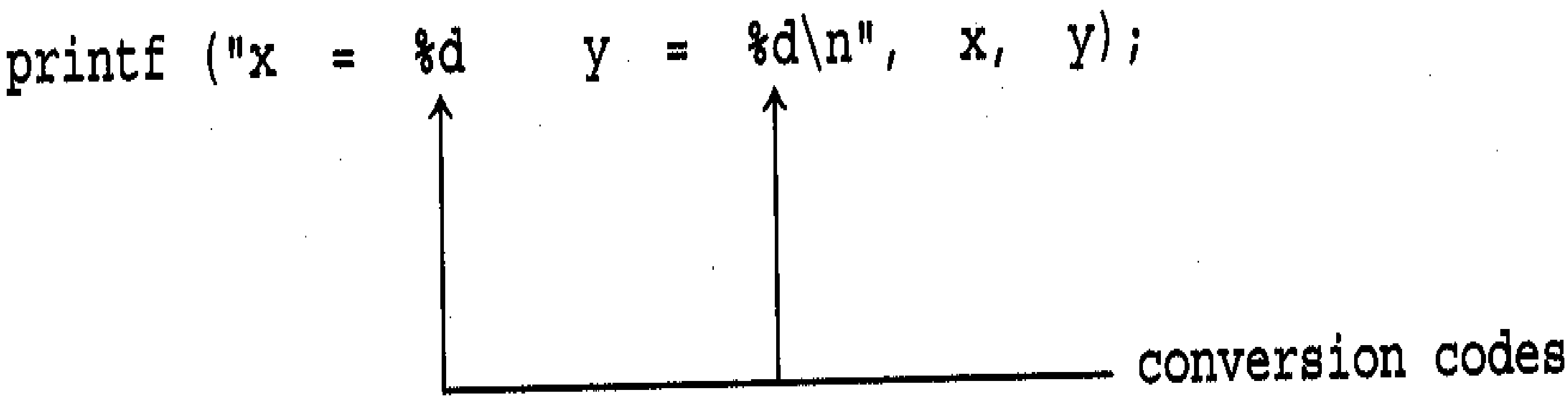
(continued)

Would not work if arguments were pushed left to right. The compiler would not know where count is.

FIGURE 9.18 c)
(continued)



printf uses variable-length argument list—the number of conversion codes indicates the number of additional arguments.



Macros

- A string replacement mechanism
- Replacement occurs during program translation, not at run time.
- Makes it easier to write code.
- Macros for variable-length parameter lists: va_list, va_start, va_arg, va_end.

va_arg macro

```
va_arg(p, int);
```

is replaced with the string

```
* ((int *) p)++;
```

first argument inserted here

second argument inserted here

```
va_arg(p, long)
```

which would be replaced with

```
*((long*)p)++
```

va_list(p);

replaced with

void *p;

```
va_start(p, count);
```

replaced with

```
p = &count + 1;
```

FIGURE 9.19

```
#include <cstdarg>
#include <iostream>
using namespace std;

int add(int count, ...)
{
      int i, sum = 0;
      va_list(p);                                    // void *p;

      va_start(p, count);                            // p = &count + 1;

      for (i = 1; i <= count; i++)
         sum = sum + va_arg(p, int);                 // sum = sum + *((int*)p)++;

      va_end(p);                                     // does nothing
      return sum;
}
void main()
{
      cout << add(3, 4, 5, 6) << endl;               // outputs 15
      cout << add(2, -10, 20) << endl;               // outputs 10
      cout << add(1, 7) << endl;                     // outputs 7
}
```